



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Electronics 2

by

Chris Stonelake

Lecturer: Dr Gary Jones

Internal Verifier: Dr. Stephen Mosey

Assignment Title

Design of an unsigned two number, uneven parallel multiplier and the design of a FSM in hardware and VHDL

30/04/2021

Abstract

Within this assignment document to main purpose is to show how a Braun multiplier is designed and simulated via the use of the Xilinx design software. And as task 2 to develop and design and finite state machine.

Digital multipliers have various formulas available some of the most efficient and common formulas are reviewed and studied in this document, industry are continually striving for increased performances whilst reducing consumed space and power for system operations for electronics that use semiconductors.

Contents

Contents.....	Page 3
Table of Figures.....	Page 4
Introduction.....	Page 5
Literature Review.....	Pages 6 - 12
Task 1 – Half adder design code.....	Pages 13 – 14
Task 1 – Half adder wave forms.....	pages 15 - 16
Task 1 – Full adder design code.....	Pages 17 - 19
Task 1 – Full adder wave form.....	Pages 20 - 21
Task 1 – Line0 Behavioural code.....	Pages 21-23
Task 1 – Line 0 rest Bench Code.....	Pages 24 - 25
Task 1 – Line0 Wave forms.....	Page 26
Task 1 – LineX Code.....	Pages 27 - 29
Task 1 – LineX Simulation and Wave Form.....	Pages 30 -31
Task 1 – Task 1 full system Braun Multiplier Diagram.....	Page 32
Task 2 – Overview and brief.....	Page 33
Task 2 – Current State and Next State Tables.....	Page 34
Task 2 – Karnaugh Maps.....	Page 35
References.....	Page 36

Table of Figures

Figure 1 – Example Braun Multiplier.....	Page 8
Figure 2 – Booth Multiplier Example.....	Page 9
Figure 3 – Wallace Tree Multiplier Example.....	Page 10
Figure 4 – VHDL code half adder circuit.....	Page 13
Figure 5 - Test bench code half adder.....	Page 14
Figure 6 - Half adder wave form when A=0/B=0.....	Page 15
Figure 7 – Changes on behavioural code B=0.....	Page 15
Figure 8 – Half adder wave form when A=0/B=1.....	Page 16
Figure 9 - Changes on behavioural code A=1 /B.....	Page 16
Figure 10 – Half adder wave form when A=1 / B=0.....	Page 16
Figure 11 – Changes on behavioural code A=1 / B=1.....	Page 16
Figure 12 – Half adder wave form when A=1 / B=1.....	Page 16
Figure 13 - Half adder truth table.....	Page 17
Figure 14 - VHDL code full adder.....	Page 18
Figure 15 - VHDL test bench.....	Page 19
Figure 16 – Full adder truth table.....	Page 20
Figure 17 - Full adder wave form.....	Page 20
Figure 18 - Full adder wave form.....	Page 21
Figure 19 – Line0 behavioural code.....	Page 22
Figure 20 - Line0 behavioural code.....	Page 23
Figure 21 – Line0 Test bench code.....	Page 24
Figure 22 – Line 0 test bench code.....	Page 25
Figure 23 – Line0 wave form.....	Page 26
Figure 24 – Line0 wave form.....	Page 26
Figure 25 – Line X integration of half adders into unit.....	Page 27
Figure 26 – LineX code.....	Page 28
Figure 27 – Line X code.....	Page 29
Figure 28 – LineX code.....	Page 29
Figure 29 – LineX code.....	Page 29
Figure 30 – LineX code.....	Page 29
Figure 31 – LineX virtual buses.....	Page 30
Figure 32 – Line X wave form.....	Page 31
Figure 33 – System over view Braun multiplier.....	Page 32
Figure 34 – State diagram as per brief.....	Page 33
Figure 35 – next and current state tables.....	Pages 34

Introduction

Within this document the attention is on the use of VHDL for the design & simulation of logic circuits. As per the assignment brief there are two types of logic circuits that are developed in this document which are as follows (digital multiplier and a finite state machine FSM).

This document reviews the most commonly used types of digital multipliers used today in literature review section, following the literature review the document will go into task 1 for designing and simulation of a 4x12 Braun multiplier.

The final section of this document will be on the task 2 section of the assignment brief, and to design and simulate the FSM using VHDL language which is then used for the code.

Literature Review

Binary multipliers are electronic circuits used in most digital circuits, this can be anything from laptops, mobile phones to simple processes such as vending machines.

Multipliers have a vital role within the digital electronic world and digital signal processing applications. Digital signalling applications can use multipliers to execute arithmetic operations such as digital filtering, digital communications and spectral analysis.

Up to the late 1970's most minicomputers did not have a multiply instruction so programmers used 'multiply routine', only after the 1978 did the industry start using multiply instruction which became common with the 16-bit generation. [1]

As more transistors per chip became available due to larger-scale integration, it became possible to put enough adders on a single chip to sum all the partial products at once, rather than reuse a single adder to handle each partial product one at a time.

Some standard digital signal processing algorithms spend most of their time multiplying, digital signal processor designers sacrifice considerable chip area in order to make the multiplier as fast as possible, a single cycle multiply-accumulate unit often used most of the chip area of early DSPs. [2]

Computers spend most of their time executing time multipliers, it is for this reason there is always a formidable appetite for faster multipliers. To have a multiplier with low power consumption is an important factor when specifying a design of a multiplier. One common way is to reduce the number of operations the device is able to execute this will then reduce the power consumption by reducing the multipliers dynamic power, that can be the greatest factor of power consumption. The goal is to ultimately produce a multiplier that packs the most much while consuming less power to deliver that extra punch in performance.

Types Of Multipliers

Within the digital multiplier world currently there are the following variations of multipliers;

- Booth Multiplier
- Combinational Multiplier
- Wallace tree Multiplier
- Array Multiplier
- Sequential Multiplier

The digital multipliers also have different types (algorithms) which are defined as the below;

Serial-Parallel Multiplier

Serial multiplication algorithm are sequential circuits with feedback. Parallel multiplication algorithm may use combinational circuits but no feedback. This type uses succeeding addition algorithm. Both operands are inputted in serial form resulting in a simple structure.

Serial Multiplier

Parallel multipliers require less hardware to do the same computations as serial multipliers and thus use less power. Where power is an important factor parallel multiplier are used more often than serial multipliers.

Parallel Multiplier

These multipliers have the speed of a parallel multiplier and the small area of a serial multiplier. They work by entering the operand serially and storing the other in parallel.

An efficient multiplier should have the following characteristics;

- Accuracy: a good multiplier should give correct results and have a high reliability
- Speed: the multiplier should perform operation at high speed.
- Area: A multiplier should occupied less number of slices.
- Power: a good multiplier should be able to give good performance for as little power as possible.
-

On the next below sections is a brief over view and visual table giving an overview of the different multipliers.

Multiplier	Performance	Complexity	Arrangement	Space
Braun Multiplier	Slow	Low	Regular	Minimal
Booth Multiplier	Average	High	Irregular	Average
Wallace Multiplier	Quick	Medium	Higly Irregular	large

Braun Multipliers

It is a simple parallel multiplier generally called as carry array multiplier, it has been restricted to perform signed bits. The structure consists of array and AND gats and adders arranged in the iterative manner and no need of logic registers. This can be called as non-addictive multipliers.

The brain multipliers was invented by Braun Edward Louis in 1963.

If the number of bits increases then the number of adders and AND gates components also increase, which may result increases the power consumption, occupies more area, take more time. Hence suitable for only less than 16 bit. Also '1' is dependant on the delay of the full adder cell and also a final adder in the last row.

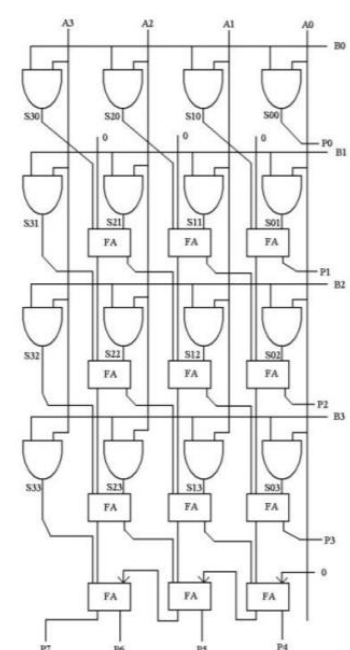


Figure 1 – Braun Multiplier Example

Booth Multiplier

Booth algorithm gives procedure for multiplying binary integrator in signed 2's complement representation in efficient way, such as less number of additions vs subtractions required. The booth multiplier operates on the premise that strings of 0's in the multiplier shall require no addition just shifting a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^k ($k=1$) to 2^m . [3]

As in all multiplications schemes, booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged.

Some advantages of the booth multiplier can be as it reduces the number of partial products generated. It can operate with signed numbers. It is well suited too work with machines tha can shoft faster than adding them.

The booth multiplier was invented by Andrew Booth in 1951.

=====									
a	1	0	1	1	0				
x	1	0	1	0	1	Multiplier			
y	-1	1	-1	1	-1	Booth-recoded			
=====									
$p^{(0)}$	0	0	0	0	0				
$+y_0a$	0	1	0	1	0				
$2p^{(1)}$	0	0	1	0	1	0			
$p^{(1)}$	0	0	1	0	1	0			
$+y_1a$	1	0	1	1	0				
$2p^{(2)}$	1	1	1	0	1	1	0		
$p^{(2)}$	1	1	1	0	1	1	0		
$+y_2a$	0	1	0	1	0				
$2p^{(3)}$	0	0	0	1	1	1	1	0	
$p^{(3)}$	0	0	0	1	1	1	1	0	
$+y_3a$	1	0	1	1	0				
$2p^{(4)}$	1	1	1	0	0	1	1	1	0
$p^{(4)}$	1	1	1	0	0	1	1	1	0
$+y_4a$	0	1	0	1	0				
$2p^{(5)}$	0	0	0	1	1	0	1	1	1
$p^{(5)}$	0	0	0	1	1	0	1	1	1
=====									

Figure 2 – Booth Multiplier Example

Wallace Tree Multiplier

Wallace multiplier is an effective parallel multiplier the first step is to form partial product array of N^2 bits. In the second step, groups of three adjacent rows each, is collected. Each group of the three rows is reduced by using full adders and half adders. Full adders are used in each Column where there are three bits, half adders are used in each Column where there are two bits. [4]

The Wallace multiplier can reduce time delay by implementing a structure that does the addition operations in parallel. We can see advantages of the multiplier in that the speed is extremely fast. By using the addition operation in parallel, reduces delay.

Wallace tree multiplier was developed by Chris Wallace in 1964.

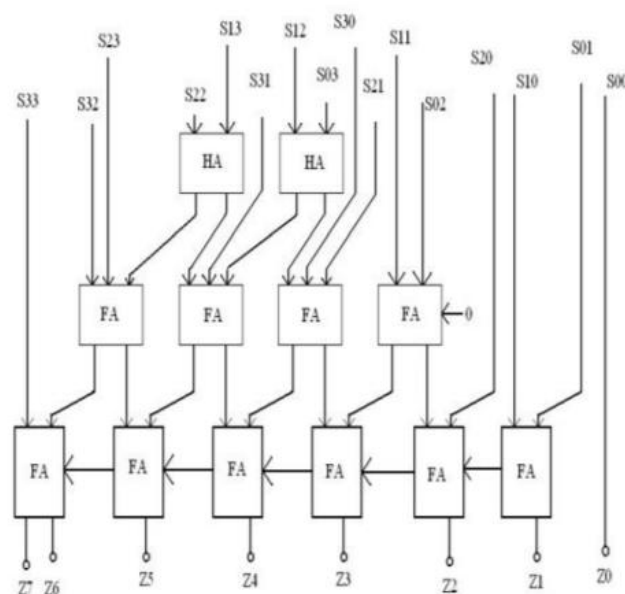


Figure 3 – Wallace Tree Multiplier Example

Conclusion from research

From the research into the above mentioned multipliers it is evident that each different type of multiplier has its role within specific projects, all have pro's and cons which is detailed in the above research but to summarise; The Braun multiplier has good performance at low power but has limitations to 16-bit operands and unsigned numbers. The booth multiplier is well suited to large operations and it reduces partial products by half while also maintaining power levels like the Braun. The Wallace tree is the fastest, but irregular structure makes designs irregular and difficult due to these factors it also inherently requires a large area.

Design over view

The 12 Bit x 4 Bit multiplier will be constructed using five different VHDL modules, the first 4 modules will be the sub modules that are required to be built first as they contain the building blocks of the multiplier. The fifth module is the main module (12X4 Bit) that will combine them into a functioning multiplier.

- VDHL Module 1 - Half Adder this is one of the two circuits which are the main body of the multiplier and the building blocks of the multiplier once the written the half adder can be used multiple times.
- VDHL Module 2 – Full Adder this is the second of the two circuits the other being the (half adder) which make up the main body multiplier and as above can be used multiple times.
- VDHL Module 3 - Multi12x4 Line 0, this is the first line at the top of the multiplier that will consist of AND gates. The signals between the gates is done with port mapping.

- VHDL Module 4 – Multi12x4line X, this circuit takes 12 of the adder circuits that are reusable and uses them as component in this line to produce a line with 8 adder circuits. The port mapping of the signals between the adders is also defined in this module.
- VHDL Module 5 – Multi12x4, the construction the 12bit x 4bit Braun multiplier can only take place once all modules 1-4 have been designed and tested. Upon completion of the 1-4 modules they can be now brought into the main module and all the inputs/out and signals are directed to the correct place in the multiplier via comprehensive port mapping, again as before this final module is tested for working functionality.

Testing Simulation

VHDL circuits designed for the multiplier maybe able to pass test and be represented in logic expression. There is no guarantee that it will function correctly. In order to ensure each VHDL modules functionality they need to be simulated, to put the design under test. Xilinx program provides a function that can put the function under test.

Each of the five VHDL test will have individual test benches righten, which will run under test at certain ports and times to create visual verification on simulation waveforms which will help as a tool to validate working order.

VDHL Module 1

Half adder circuit can add two single binary digits A&B. the half adder has two outputs, suma and carry-out. The carry out can send out a signal into an additional part of a circuit. It has no carry input and so it is only to be used in the circuit where there is no carry in signal to slightly reduce hardware.

The half adder VHDL code as per (below image 4) is the half adder circuit.

Half adder VHDL code

The VHDL code below in figure 4 is of the half adder circuit, looking at lines 20 to 47 code is written to first list the entity declarations such as name and list if it is an input or output. Entity behaviour

```
19 -----
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 -- Uncomment the following library declaration if using
24 -- arithmetic functions with Signed or Unsigned values
25 --use IEEE.NUMERIC_STD.ALL;
26
27 -- Uncomment the following library declaration if instantiating
28 -- any Xilinx primitives in this code.
29 --library UNISIM;
30 --use UNISIM.VComponents.all;
31
32
33 entity halfadder is
34     Port ( A : in  STD_LOGIC;      -- A is set as an input
35           B : in  STD_LOGIC;      -- B is set as an input
36           Sum : out STD_LOGIC;    -- Sum is set as an output
37           Carry : out STD_LOGIC); -- Carry is set as an output
38 end halfadder;
39
40 architecture Behavioral of halfadder is
41
42 begin
43
44     Sum <= A xor B; -- Sum is assigned A and B an xor gate
45     Carry <= A and B; -- Carry is assigned A and B an And gate
46
47 end Behavioral;
48
49
```

Figure 4 – VHDL code half adder circuit

Following the process one the logic circuit has been completed written in the VHDL and test. The VHDL test bench file was created in figure 5 as below.

Lines 65 to 86 of the test bench code are to generate stimuli on the inputs ports A and B by using the command `stim_proc` to switch the input states from 0 to 1. Input A is switched every 10ns (as shown on line 77 of figure 5). And input B is switched every 20ns (again shown in figure 5, line 85). This will result in a signal waveform pattern where the internal input signals will cross over at different points with different states to simulate all the different states that the half adder could cycle through.

```

27 -----
28 LIBRARY ieee;
29 USE ieee.std_logic_1164.ALL;
30
31 -- Uncomment the following library declaration if using
32 -- arithmetic functions with Signed or Unsigned values
33 --USE ieee.numeric_std.ALL;
34
35 ENTITY halfadder_TB IS
36 END halfadder_TB;
37
38 ARCHITECTURE behavior OF halfadder_TB IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT halfadder
43     PORT (
44         A : IN std_logic;
45         B : IN std_logic;
46         Sum : OUT std_logic;
47         Carry : OUT std_logic
48     );
49     END COMPONENT;
50
51
52     --Inputs
53     signal A : std_logic := '0';
54     signal B : std_logic := '0';
55
56     --Outputs
57     signal Sum : std_logic;
58     signal Carry : std_logic;
59     -- No clocks detected in port list. Replace <clock> below with
60     -- appropriate port name
61
62 BEGIN
63
64     -- Instantiate the Unit Under Test (UUT)
65     uut: halfadder PORT MAP (
66         A => A,
67         B => B,
68         Sum => Sum,
69         Carry => Carry
70     );
71
72     -- Stimulus process
73     stim_proc0: process
74     begin
75         -- hold reset state for 100 ns.
76         wait for 10 ns;
77         A <= not A;
78
79     end process;
80
81     stim_proc1: process
82     begin
83         -- hold reset state for 100 ns.
84         wait for 20 ns;
85         B <= not B;
86
87     end process;
88
89 END;
90

```

Figure 5 – test bench code half adder

The behaviour model simulator can be used to visually confirm that the design is functioning as designed. The results of the stimulus can be seen with input A switching low to high every 10ns and input B switching low to high every 20ns the half adders wave form can be seen in the below figure 6.

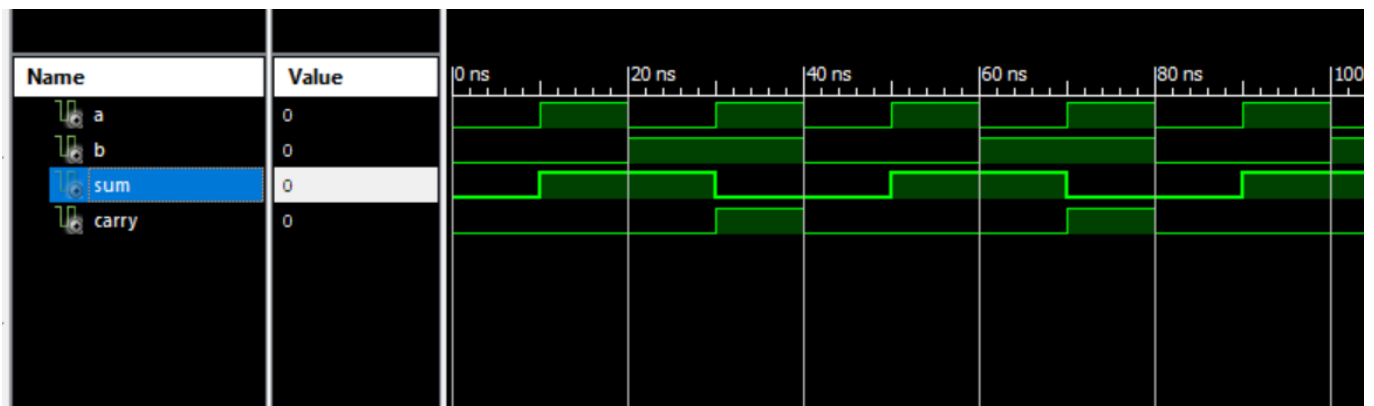


Figure 6 – Half Adder Wave Form _ when A=0 and B=0

The next figures 7&8 show the half adder model where A=0 and B=1.

```

51
52  --Inputs
53  signal A : std_logic := '0';
54  signal B : std_logic := '1';
55

```

Figure 7 _ changes on behavioural model B=0

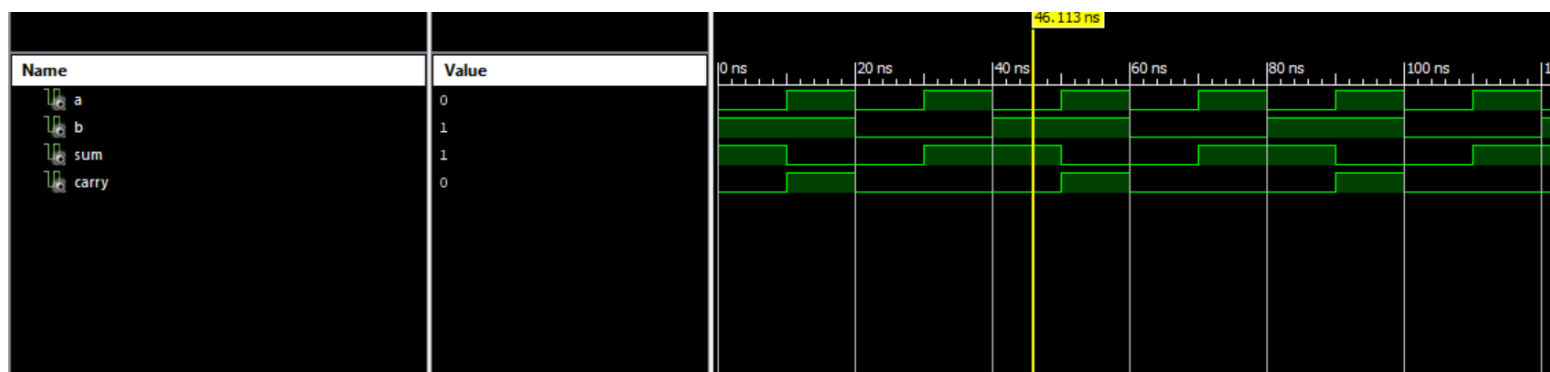


Figure 8 _ Half Adder Wave Form _ When A=0 and B=1

The next figures 9&10 show the half adder model and wave form where A=1 & B=0

```

52  --Inputs
53  signal A : std_logic := '1';
54  signal B : std_logic := '0';
55

```

Figure 9 _ changes on behavioural model A=1 and B=0

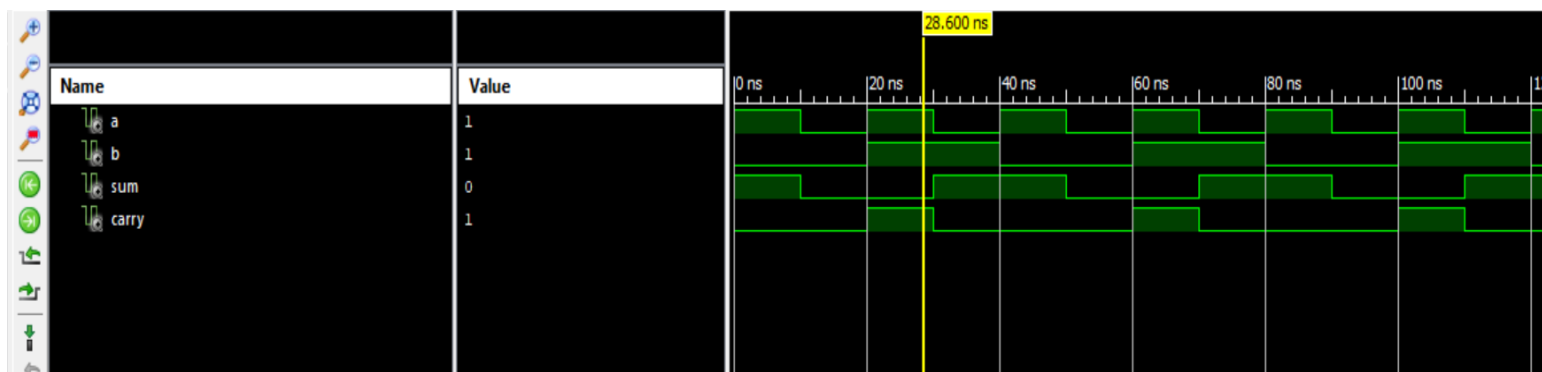


Figure 10 _ Half adder wave form _ When A=1 and B=0

The next figures 11 & 12 show the half adder model and wave form when A=1 & B=1

```

52  --Inputs
53  signal A : std_logic := '1';
54  signal B : std_logic := '1';
55

```

Figure 11 _ changes on behavioural model A=1 & B=1

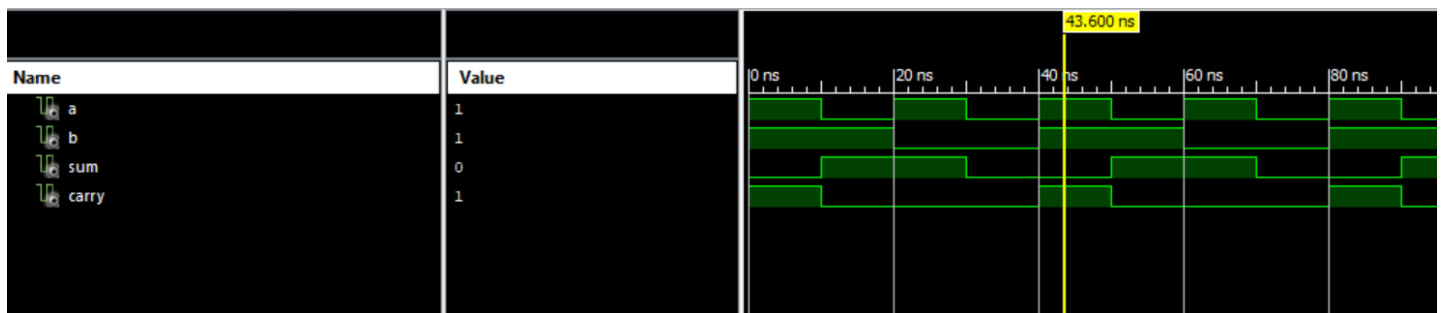


Figure 12 half adder wave form _ when A=1 and B=1

Based on the results shown in the half adder figures 6 to 12 all 4 possible states of the half adder the simulated test show that the half adder circuit is functioning correctly in line with the half adders truth table as shown below in figure 12.

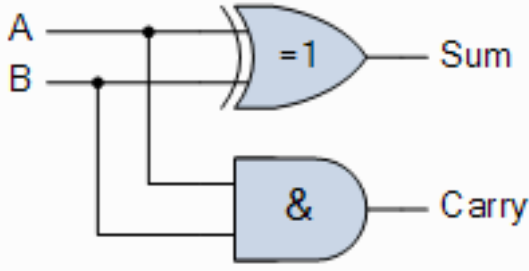
Symbol	Truth Table			
	B	A	SUM	CARRY
	0	0	0	0
	0	1	1	0
	1	0	1	0
	1	1	0	1

Figure 13 _ Half adder truth table

Full Adder VHDL code

The full adder is as expected different from the half adder as it has a carry in port (C) and three more logic gates (OR, AND, XOR) the extra port is required, and the architecture require extra code to state the initial signals and the functions of the circuit.

Full adder VHDL code

for the full adder VHDL code in figure 13is of the full adder circuit. At lines 33 to 39 this code is written to first list the entity declarations such as entity name and list if the entity is an inout or output.

Code on lines 41 to 53 show the architecture body that describes the relationship between the ports by using the entity to state what other entity it is interconnected to and by what type of logic gate signals are connected to which port, just as they are in the full adder truth table (figure 14).

```

19 -----
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 -- Uncomment the following library declaration if using
24 -- arithmetic functions with Signed or Unsigned values
25 --use IEEE.NUMERIC_STD.ALL;
26
27 -- Uncomment the following library declaration if instantiating
28 -- any Xilinx primitives in this code.
29 --library UNISIM;
30 --use UNISIM.VComponents.all;
31
32
33 entity fulladder is
34     Port ( A : in  STD_LOGIC;      -- A is set as an input
35           B : in  STD_LOGIC;      -- B is set as an input
36           C : in  STD_LOGIC;      -- Carry (in) is set as an input
37           Sum : out STD_LOGIC;     -- Sum is set as an output
38           Carry : out STD_LOGIC); -- Carry (out) is set as an onput
39 end fulladder;
40
41 architecture Behavioral of fulladder is
42
43     signal AxB : STD_LOGIC; -- logic set to signals
44     signal AxBnC: STD_LOGIC; -- logic set to signals
45     signal AnB : STD_LOGIC; -- logic set to signals
46
47 begin
48     AxB    <= A xor B;
49     AxBnC <= AxB and C;
50     AnB    <= A and B;
51
52     Sum    <= AxB xor C;
53     Carry  <= AxBnC or AnB;
54
55 end Behavioral;
56
57

```

Figure 14 _ VHDL code full adder circuit

With the VHDL designed, attest bench was created shown below in figure 14, this was to test and verify the design and works accordingly. The code was made to source input in that the circuits can respond and generate stimuli on the inputs A,B and C by using the command stim_proc to switch these inputs at different times from 0 to 1 repeatably.

Input A is switched every 10ns, input B is switched every 20ns, and input C is switched every 40ns. This created patters that show the signals crossing over at different stages which shows all the different states the full adder can cycle through.

```

27 -----
28 LIBRARY ieee;
29 USE ieee.std_logic_1164.ALL;
30
31 -- Uncomment the following library declaration if using
32 -- arithmetic functions with Signed or Unsigned values
33 --USE ieee.numeric_std.ALL;
34
35 ENTITY fulladder_TB IS
36 END fulladder_TB;
37
38
39
40 ARCHITECTURE behavior OF fulladder_TB IS
41
42     -- Component Declaration for the Unit Under Test (UUT)
43
44     COMPONENT fulladder
45     PORT(
46         A : IN  std_logic;
47         B : IN  std_logic;
48         C : IN  std_logic;
49         Sum : OUT std_logic;
50         Carry : OUT std_logic
51     );
52     END COMPONENT;
53
54     --Inputs
55     signal A : std_logic := '0';
56     signal B : std_logic := '0';
57     signal C : std_logic := '0';
58
59     --Outputs
60     signal Sum : std_logic;
61     signal Carry : std_logic;
62
63
64 BEGIN
65     -- Instantiate the Unit Under Test (UUT)
66     uut: fulladder PORT MAP (
67         A => A,
68         B => B,
69         C => C,
70         Sum => Sum,
71         Carry => Carry
72     );
73
74     -- Stimulus process
75     stim_proc: process
76     begin
77         -- hold reset state for 10 ns.
78         wait for 10 ns;
79         A <= not A;
80     end process;
81
82     stim_proc1: process
83     begin
84         -- hold reset state for 20 ns.
85         wait for 20 ns;
86         B <= not B;
87     end process;
88
89     stim_proc2: process
90     begin
91         -- hold reset state for 40 ns.
92         wait for 40 ns;
93         C <= not C;
94     end process;
95
96 END;

```

Figure 15 – VHDL test bench

With the full adder truth table as below (figure 14) the implementation of a full adder circuit can be understood easily. The sum 'S' is produced in two steps.

1. By XORing the provided inputs 'A' and 'B'
2. The result of A XOR B is then XORed with the C-IN

This generates SUM and C-OUT is true only when either two of three inputs are High, the C-out will also be high. Implementation of the full adder circuit with the help of the half adder circuits. Initially, the half adder will be used to add A and B to produce by the first half adder to get the final S output.

INPUTS			OUTPUT	
A	B	C-IN	C-OUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 16 – Full adder truth table.

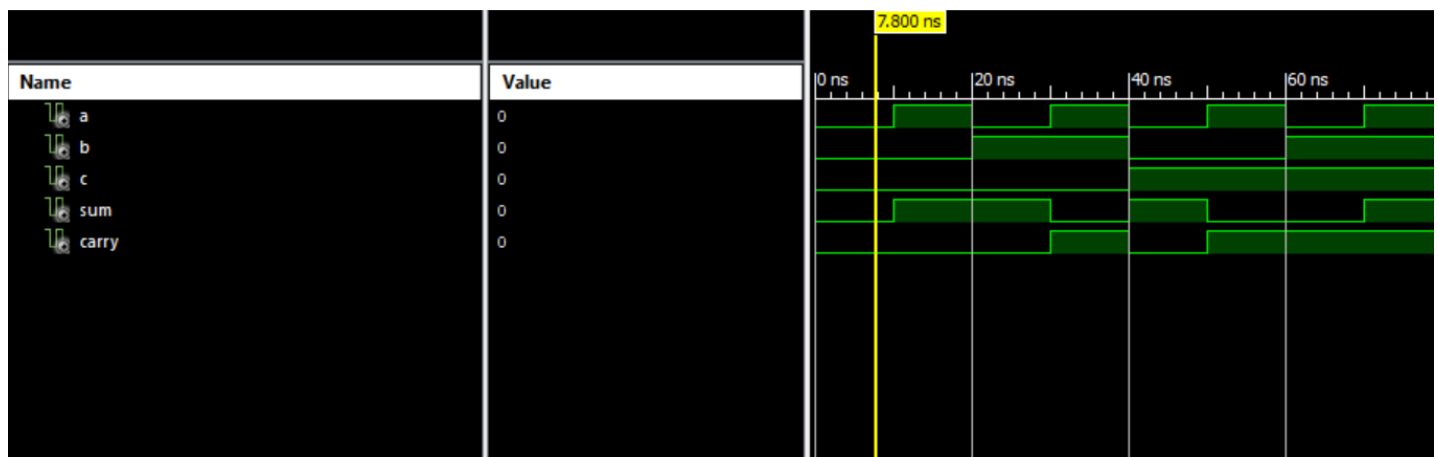


Figure 17 – Full adder wave form

As shown in the above figure 15, for the full adder wave form the in in-puts and out-puts are inline with the full adder truth table in line 1 of figure 14.

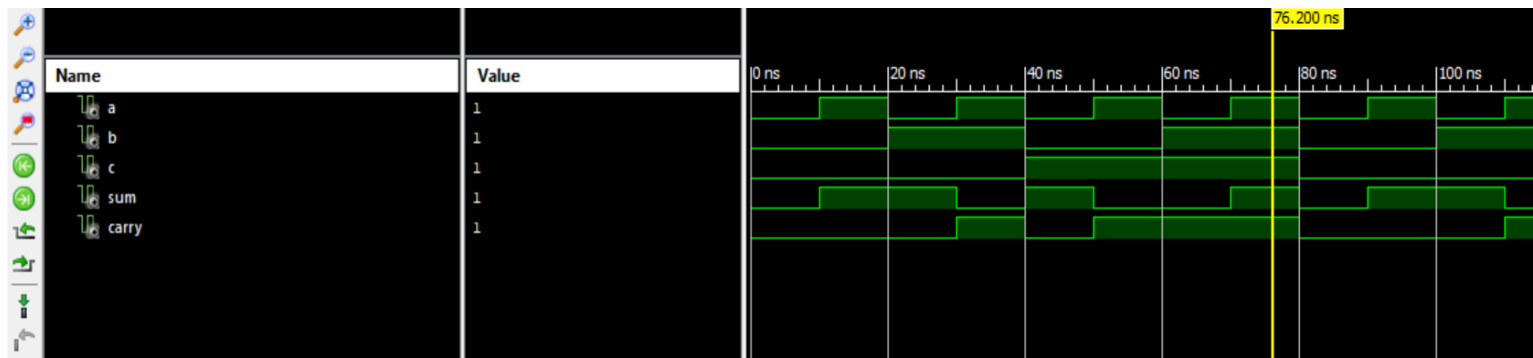


Figure 18 – Fuller adder wave form

Again as shown in the above figure 16, for the full adder wave form the in-puts and out-puts are inline wit the full adder truth table in line 8 of figure 14.

With the above figures showing the full adder all possible states have been proven and show that the half adder input is functioning in line with the full adder truth table.

Based on the above result it is sound to state that the circuits have performed in line with expectations, using the test bench to write code that was able to generate inputs for ports A,B and C.

Line 0

Within this next step the creation of the line 0 beaviourial code, this is shown in figure 17. In figure 17 the code shows for lines 32 to 59 indicate the entity declarations naming A1 to A12 as in-put ports and Q1 to Q11 as out-put ports.

Code shown on figure 17 line 64 to 75 shows the architecture behaviour of line 0, and describes the relationship between ports.

```

19 -----
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 -- Uncomment the following library declaration if using
24 -- arithmetic functions with Signed or Unsigned values
25 --use IEEE.NUMERIC_STD.ALL;
26
27 -- Uncomment the following library declaration if instantiating
28 -- any Xilinx primitives in this code.
29 --library UNISIM;
30 --use UNISIM.VComponents.all;
31
32 entity line0 is
33     Port ( A1 : in  STD_LOGIC;
34           A2 : in  STD_LOGIC;
35           A3 : in  STD_LOGIC;
36           A4 : in  STD_LOGIC;
37           A5 : in  STD_LOGIC;
38           A6 : in  STD_LOGIC;
39           A7 : in  STD_LOGIC;
40           A8 : in  STD_LOGIC;
41           A9 : in  STD_LOGIC;
42           A10 : in STD_LOGIC;
43           A11 : in STD_LOGIC;
44           A12 : in STD_LOGIC;
45           B1 : in  STD_LOGIC;
46           Q0 : out STD_LOGIC;
47           Q1 : out STD_LOGIC;
48           Q2 : out STD_LOGIC;
49           Q3 : out STD_LOGIC;
50           Q4 : out STD_LOGIC;
51           Q5 : out STD_LOGIC;
52           Q6 : out STD_LOGIC;
53           Q7 : out STD_LOGIC;
54           Q8 : out STD_LOGIC;
55           Q9 : out STD_LOGIC;
56           Q10 : out STD_LOGIC;
57           Q11 : out STD_LOGIC);
58 end line0;
59
60 architecture Behavioral of line0 is
61
62 begin
63
64     Q0 <= A1 and B1;
65     Q1 <= A2 and B1;
66     Q2 <= A3 and B1;
67     Q3 <= A4 and B1;
68     Q4 <= A5 and B1;
69     Q5 <= A6 and B1;
70     Q6 <= A7 and B1;
71     Q7 <= A8 and B1;
72     Q8 <= A9 and B1;
73     Q9 <= A10 and B1;
74     Q10 <= A11 and B1;
75     Q11 <= A12 and B1;
76
77 end Behavioral;
78
79

```

Figure 19 – Line 0 Behavioural code

Line0 test bench code

Now that the Line0 VHDL code has been made as per figure 17, a VDHL has been created to verify the design under test.

The test bench code will generate inputs so that the circuit can react and generate outputs in the simulator.

As per figure 18 below, which shows the test bench code, lines from 74 to 86 of the test bench are to create stimulus on the In-puts A1 to A12 and B1.

```

38 ARCHITECTURE behavior OF line_tb IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT line0
43     PORT(
44         A1 : IN  std_logic;
45         A2 : IN  std_logic;
46         A3 : IN  std_logic;
47         A4 : IN  std_logic;
48         A5 : IN  std_logic;
49         A6 : IN  std_logic;
50         A7 : IN  std_logic;
51         A8 : IN  std_logic;
52         A9 : IN  std_logic;
53         A10 : IN  std_logic;
54         A11 : IN  std_logic;
55         A12 : IN  std_logic;
56         B1 : IN  std_logic;
57         Q0 : OUT std_logic;
58         Q1 : OUT std_logic;
59         Q2 : OUT std_logic;
60         Q3 : OUT std_logic;
61         Q4 : OUT std_logic;
62         Q5 : OUT std_logic;
63         Q6 : OUT std_logic;
64         Q7 : OUT std_logic;
65         Q8 : OUT std_logic;
66         Q9 : OUT std_logic;
67         Q10 : OUT std_logic;
68         Q11 : OUT std_logic;
69     );
70     END COMPONENT;
71
72     --Inputs
73     signal A1 : std_logic := '0';
74     signal A2 : std_logic := '0';
75     signal A3 : std_logic := '0';
76     signal A4 : std_logic := '0';
77     signal A5 : std_logic := '0';
78     signal A6 : std_logic := '0';
79     signal A7 : std_logic := '0';
80     signal A8 : std_logic := '0';
81     signal A9 : std_logic := '0';
82     signal A10 : std_logic := '0';
83     signal A11 : std_logic := '0';
84     signal A12 : std_logic := '0';
85     signal B1 : std_logic := '0';
86
87     --Outputs
88     signal Q0 : std_logic;
89     signal Q1 : std_logic;
90     signal Q2 : std_logic;
91     signal Q3 : std_logic;
92     signal Q4 : std_logic;
93     signal Q5 : std_logic;
94     signal Q6 : std_logic;
95     signal Q7 : std_logic;
96     signal Q8 : std_logic;
97     signal Q9 : std_logic;
98     signal Q10 : std_logic;
99     signal Q11 : std_logic;
100
101
102 BEGIN
103
104

```

Figure 20_ Line 0 behaviour code

```

102
103 BEGIN
104
105     -- Instantiate the Unit Under Test (UUT)
106     uut: line0 PORT MAP (
107         A1 => A1,
108         A2 => A2,
109         A3 => A3,
110         A4 => A4,
111         A5 => A5,
112         A6 => A6,
113         A7 => A7,
114         A8 => A8,
115         A9 => A9,
116         A10 => A10,
117         A11 => A11,
118         A12 => A12,
119         B1 => B1,
120         Q0 => Q0,
121         Q1 => Q1,
122         Q2 => Q2,
123         Q3 => Q3,
124         Q4 => Q4,
125         Q5 => Q5,
126         Q6 => Q6,
127         Q7 => Q7,
128         Q8 => Q8,
129         Q9 => Q9,
130         Q10 => Q10,
131         Q11 => Q11
132     );
133
134
135     -- Stimulus process
136     stim_proc: process
137     begin
138         -- hold reset state for 100 ns.
139         wait for 1 ns;
140         A1<= not A1;
141     end process;
142
143     stim_proc2: process
144     begin
145         -- hold reset state for 100 ns.
146         wait for 2 ns;
147         A2<= not A2;
148     end process;
149
150     stim_proc3: process
151     begin
152         -- hold reset state for 100 ns.
153         wait for 4 ns;
154         A3<= not A3;
155     end process;
156
157     stim_proc4: process
158     begin
159         -- hold reset state for 100 ns.
160         wait for 8 ns;
161         A4<= not A4;
162     end process;

```

Figure 21 _ Test bench code for Line0

```

164 stim_proc5: process
165 begin
166     -- hold reset state for 100 ns.
167     wait for 16 ns;
168     A5<= not A5;
169 end process;
170
171 stim_proc6: process
172 begin
173     -- hold reset state for 100 ns.
174     wait for 32 ns;
175     A6<= not A6;
176 end process;
177
178 stim_proc7: process
179 begin
180     -- hold reset state for 100 ns.
181     wait for 64 ns;
182     A7<= not A7;
183 end process;
184
185 stim_proc8: process
186 begin
187     -- hold reset state for 100 ns.
188     wait for 128 ns;
189     A8<= not A8;
190 end process;
191
192 stim_proc9: process
193 begin
194     -- hold reset state for 100 ns.
195     wait for 128 ns;
196     A8<= not A8;
197 end process;
198
199 stim_procl0: process
200 begin
201     -- hold reset state for 100 ns.
202     wait for 256 ns;
203     A9<= not A9;
204 end process;
205
206 stim_procl1: process
207 begin
208     -- hold reset state for 100 ns.
209     wait for 512 ns;
210     A10<= not A10;
211 end process;
212
213 stim_procl2: process
214 begin
215     -- hold reset state for 100 ns.
216     wait for 1024 ns;
217     A11<= not A11;
218 end process;
219
220 stim_procl3: process
221 begin
222     -- hold reset state for 100 ns.
223     wait for 2048 ns;
224     A12<= not A12;
225 end process;
226
227 stim_procl4: process
228 begin
229     -- hold reset state for 100 ns.
230     wait for 4096 ns;
231     B1<= not B1;
232 end process;
233
234 END;
235

```

Figure 22 _ Test bench code for Line 0

As shown on figures 19 & 20 the inputs are switched in starting with A1 AT 1ns as the sequences carry's on through too line 232, the time intervals will double at each stage, at the point of line 230 the switching time is as 4096ns.

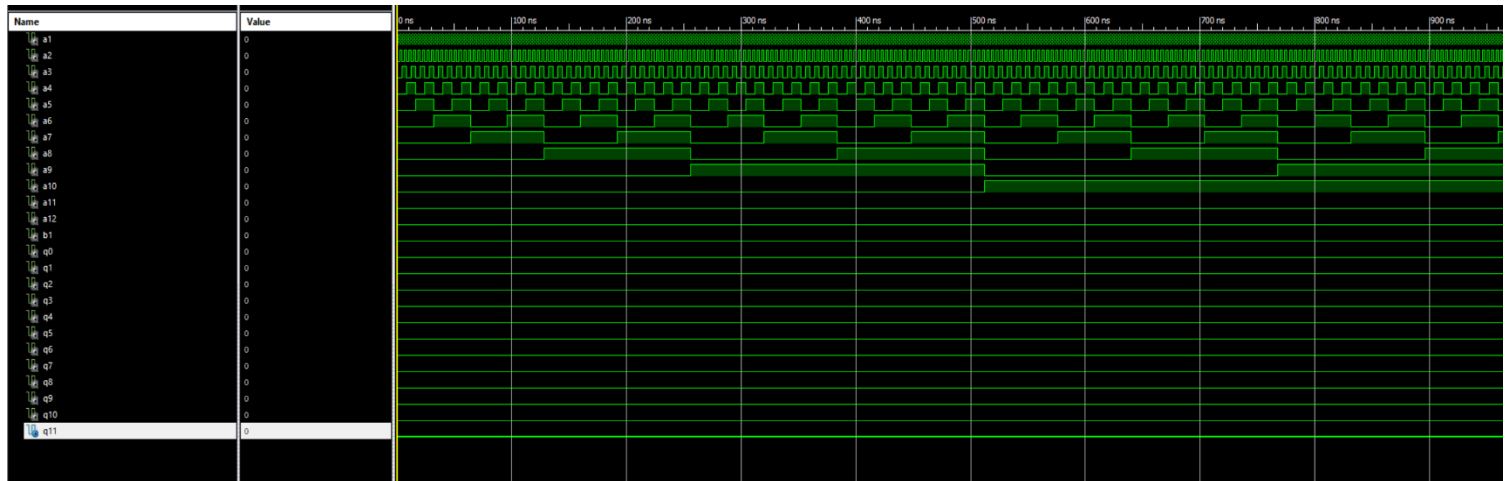


Figure 23 – Line 0 wave form

Represented as above in figure 21, the simulator is to test that all in-puts are at 0, if X is low and secondly to ensure ethe out-put valves stay equal if values X is 1. The radix has been changed to ensign decimal to help show the input and output values.

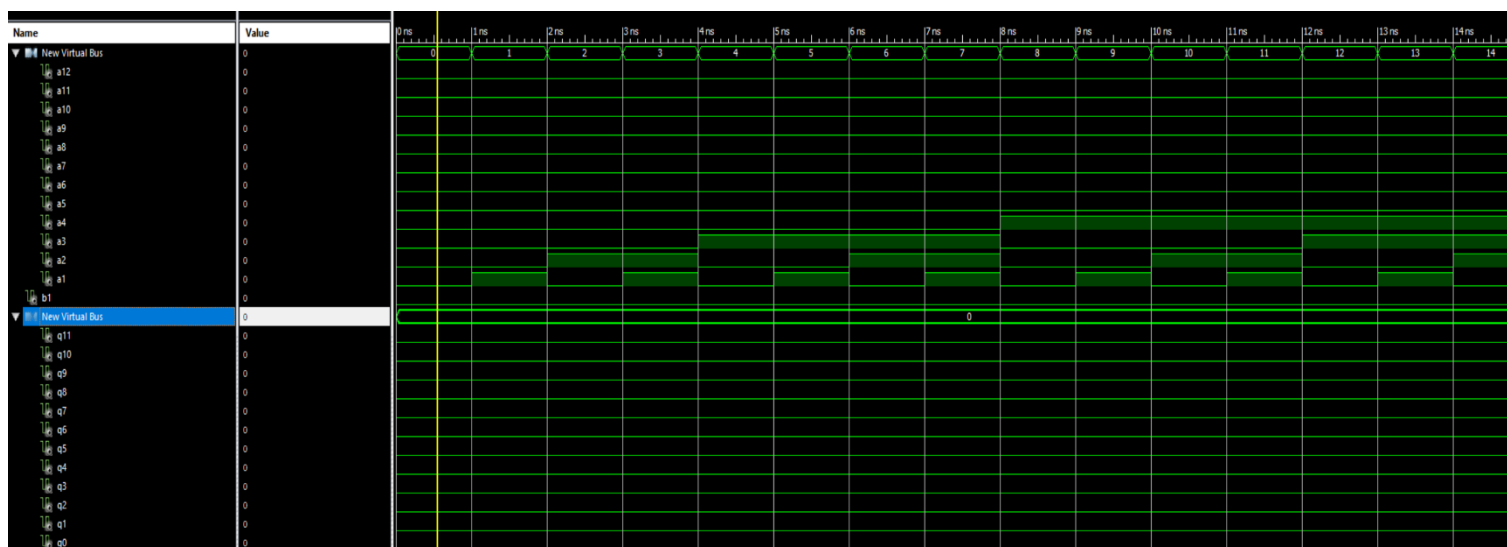


Figure 24 – Line 0 wave form

Line X Code

Within this section of the LineX VHDL module extensively uses the VHDL modules of the half adder and full adder. This is done by adding the adders modules to this section as components and specify their connections in the VHDL. All of the adders used in this unit need to have internal and external signals.

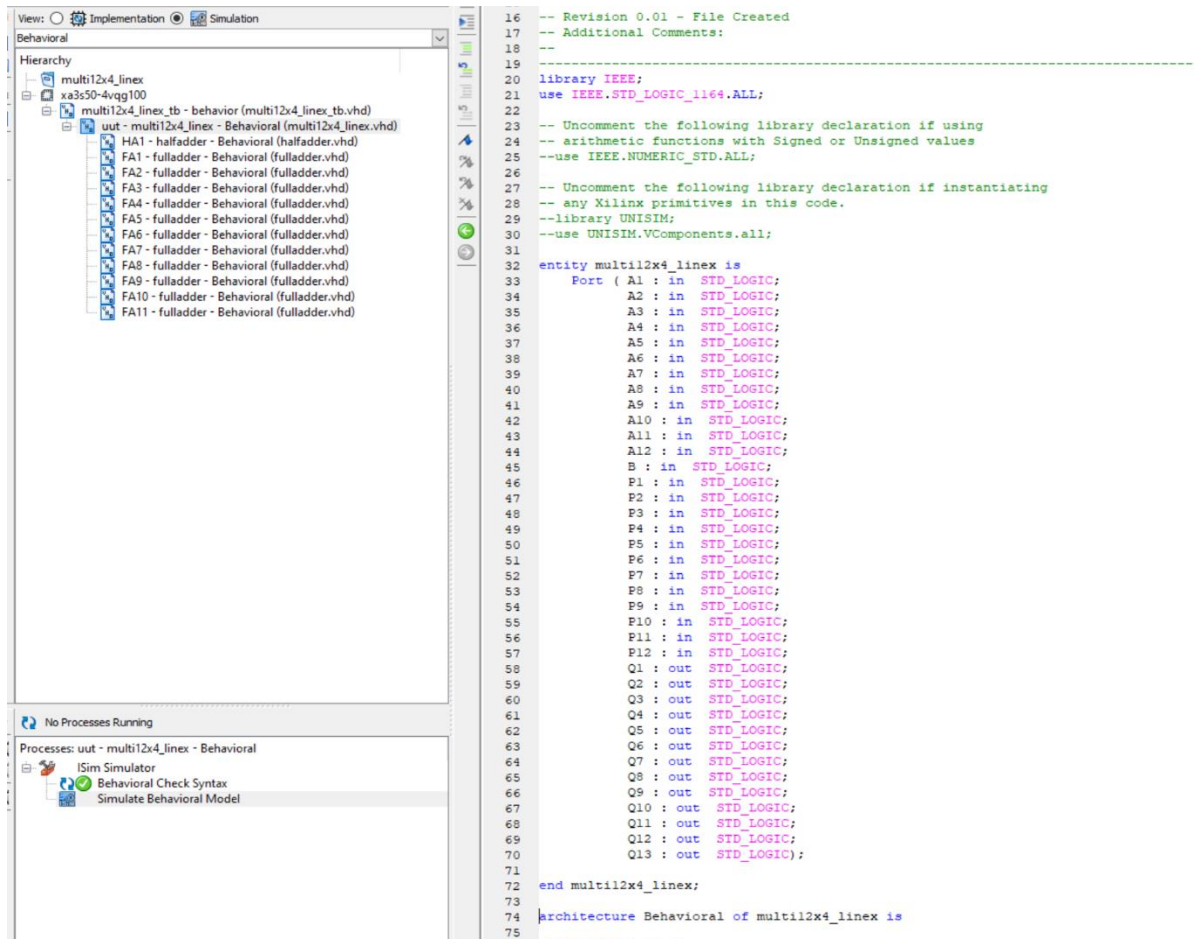


Figure 25 – Line X _ integration of half and full adders into unit

As shown in the above figure 24, the internal signals are as per lines 44 to 16 (A0 to P12). The external signals are in lines 69 to 81 (Q1 to Q13).

```

42  COMPONENT multil2x4_linex
43  PORT (
44      A1 : IN  std_logic;
45      A2 : IN  std_logic;
46      A3 : IN  std_logic;
47      A4 : IN  std_logic;
48      A5 : IN  std_logic;
49      A6 : IN  std_logic;
50      A7 : IN  std_logic;
51      A8 : IN  std_logic;
52      A9 : IN  std_logic;
53      A10 : IN  std_logic;
54      A11 : IN  std_logic;
55      A12 : IN  std_logic;
56      B : IN  std_logic;
57      P1 : IN  std_logic;
58      P2 : IN  std_logic;
59      P3 : IN  std_logic;
60      P4 : IN  std_logic;
61      P5 : IN  std_logic;
62      P6 : IN  std_logic;
63      P7 : IN  std_logic;
64      P8 : IN  std_logic;
65      P9 : IN  std_logic;
66      P10 : IN  std_logic;
67      P11 : IN  std_logic;
68      P12 : IN  std_logic;
69      Q1 : OUT std_logic;
70      Q2 : OUT std_logic;
71      Q3 : OUT std_logic;
72      Q4 : OUT std_logic;
73      Q5 : OUT std_logic;
74      Q6 : OUT std_logic;
75      Q7 : OUT std_logic;
76      Q8 : OUT std_logic;
77      Q9 : OUT std_logic;
78      Q10 : OUT std_logic;
79      Q11 : OUT std_logic;
80      Q12 : OUT std_logic;
81      Q13 : OUT std_logic;
82  );
83  END COMPONENT;
84

```

Figure 26 – Line X code

For line X on the code lines 134 to 365 are the signals that assign the integral signal connections to the correct inputs, with large port mapping. This is shown in the below figure's 25 to 28.

```

132 -- Instantiate the Unit Under Test (UUT)
133 uut: multil2x4_lindex PORT MAP (
134     A1 => A1,
135     A2 => A2,
136     A3 => A3,
137     A4 => A4,
138     A5 => A5,
139     A6 => A6,
140     A7 => A7,
141     A8 => A8,
142     A9 => A9,
143     A10 => A10,
144     A11 => A11,
145     A12 => A12,
146     B => B,
147     P1 => P1,
148     P2 => P2,
149     P3 => P3,
150     P4 => P4,
151     P5 => P5,
152     P6 => P6,
153     P7 => P7,
154     P8 => P8,
155     P9 => P9,
156     P10 => P10,
157     P11 => P11,
158     P12 => P12,
159     Q1 => Q1,
160     Q2 => Q2,
161     Q3 => Q3,
162     Q4 => Q4,
163     Q5 => Q5,
164     Q6 => Q6,
165     Q7 => Q7,
166     Q8 => Q8,
167     Q9 => Q9,
168     Q10 => Q10,
169     Q11 => Q11,
170     Q12 => Q12,
171     Q13 => Q13

```

figure 27

```

175 -- Stimulus process
176 stim_proc: process
177 begin
178     -- hold reset state for 100 ns.
179     wait for 10 ns;
180     A1 <= not A1;
181     end process;
182
183 -- Stimulus process
184 stim_proc2: process
185 begin
186     -- hold reset state for 100 ns.
187     wait for 20 ns;
188     A2 <= not A2;
189     end process;
190
191 -- Stimulus process
192 stim_proc3: process
193 begin
194     -- hold reset state for 100 ns.
195     wait for 40 ns;
196     A3 <= not A3;
197     end process;
198
199 -- Stimulus process
200 stim_proc4: process
201 begin
202     -- hold reset state for 100 ns.
203     wait for 80 ns;
204     A4 <= not A4;
205     end process;
206
207 -- Stimulus process
208 stim_proc5: process
209 begin
210     -- hold reset state for 100 ns.
211     wait for 160 ns;
212     A5 <= not A5;
213     end process;
214
215 -- Stimulus process
216 stim_proc6: process
217 begin
218     -- hold reset state for 100 ns.
219     wait for 320 ns;
220     A6 <= not A6;
221     end process;
222
223 -- Stimulus process
224 stim_proc7: process
225 begin
226     -- hold reset state for 100 ns.
227     wait for 640 ns;
228     A7 <= not A7;
229     end process;
230
231 -- Stimulus process
232 stim_proc8: process
233 begin
234     -- hold reset state for 100 ns.
235     wait for 1280 ns;
236     A8 <= not A8;
237     end process;

```

Figure 28

```

239 -- Stimulus process
240 stim_proc9: process
241 begin
242     -- hold reset state for 100 ns.
243     wait for 2560 ns;
244     A9 <= not A9;
245     end process;
246
247 -- Stimulus process
248 stim_proc10: process
249 begin
250     -- hold reset state for 100 ns.
251     wait for 5120 ns;
252     A10 <= not A10;
253     end process;
254
255 -- Stimulus process
256 stim_proc11: process
257 begin
258     -- hold reset state for 100 ns.
259     wait for 10240 ns;
260     A11 <= not A11;
261     end process;
262
263 -- Stimulus process
264 stim_proc12: process
265 begin
266     -- hold reset state for 100 ns.
267     wait for 20480 ns;
268     A12 <= not A12;
269     end process;
270
271 -- Stimulus process
272 stim_proc13: process
273 begin
274     -- hold reset state for 100 ns.
275     wait for 40960 ns;
276     P1 <= not P1;
277     end process;
278
279 -- Stimulus process
280 stim_proc14: process
281 begin
282     -- hold reset state for 100 ns.
283     wait for 81920 ns;
284     P2 <= not P2;
285     end process;
286
287 -- Stimulus process
288 stim_proc15: process
289 begin
290     -- hold reset state for 100 ns.
291     wait for 163840 ns;
292     P3 <= not P3;
293     end process;
294
295 -- Stimulus process
296 stim_proc16: process
297 begin
298     -- hold reset state for 100 ns.
299     wait for 327680 ns;
300     P4 <= not P4;
301     end process;
302

```

```

303 -- Stimulus process
304 stim_proc17: process
305 begin
306     -- hold reset state for 100 ns.
307     wait for 655360 ns;
308     P5 <= not P5;
309     end process;
310
311 -- Stimulus process
312 stim_proc18: process
313 begin
314     -- hold reset state for 100 ns.
315     wait for 1310720 ns;
316     P6 <= not P6;
317     end process;
318
319 -- Stimulus process
320 stim_proc19: process
321 begin
322     -- hold reset state for 100 ns.
323     wait for 2621440 ns;
324     P7 <= not P7;
325     end process;
326
327 -- Stimulus process
328 stim_proc20: process
329 begin
330     -- hold reset state for 100 ns.
331     wait for 5242880 ns;
332     P8 <= not P8;
333     end process;
334
335 -- Stimulus process
336 stim_proc21: process
337 begin
338     -- hold reset state for 100 ns.
339     wait for 10485760 ns;
340     P9 <= not P9;
341     end process;
342
343 -- Stimulus process
344 stim_proc22: process
345 begin
346     -- hold reset state for 100 ns.
347     wait for 20971520 ns;
348     P10 <= not P10;
349     end process;
350
351 -- Stimulus process
352 stim_proc23: process
353 begin
354     -- hold reset state for 100 ns.
355     wait for 41943040 ns;
356     P11 <= not P11;
357     end process;
358
359 -- Stimulus process
360 stim_proc24: process
361 begin
362     -- hold reset state for 100 ns.
363     wait for 83886080 ns;
364     P12 <= not P12;
365     end process;

```

Figure 29

Figure 30

Now that the lineX circuit has been completed in terms of code written in the VHDL and the syntheses check completed. A VDHL test bench file was created to test and scrutinise the code and verify its correct operation.

The test bench code seen below was created to generate stimuli so that the circuit can respond and create outputs in the simulator these are shown of figures 26 to 28.

Lines 176 to 365 are to create the test stimulus on the in-puts and by first setting them to 10ns using the commend sim protocol to switch the inputs from 0 to 1 repetably for a defined times which then keeps doubling until the last switching which has a switching time of 83886080ns.

LineX Simulation

Creation of the simulation test bench, 3 x virtual busses were created which are segregated into A, P and Q as shown below in figure 29.

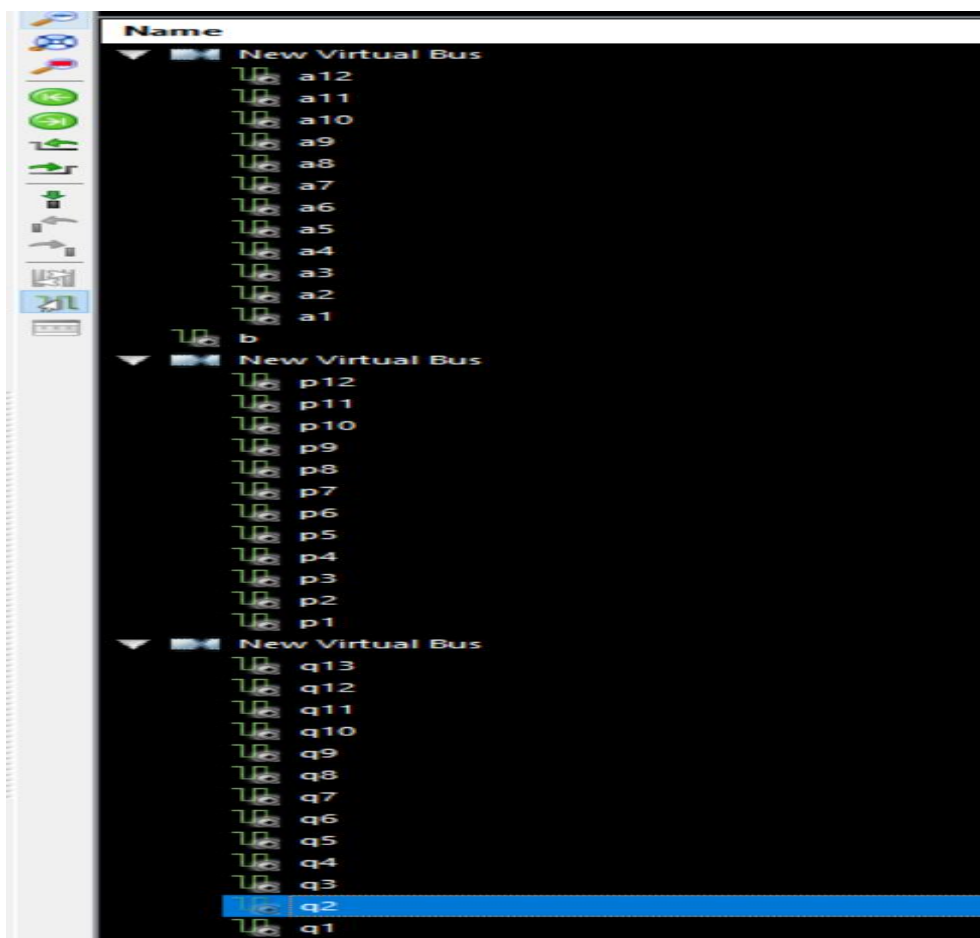


Figure 31 _line x virtual buses

The simulation for lineX should give a results so that when X is low or 0 no input values should be added to the previous value. Also, if X is high or 1 then the value in the (IN) should be added to the previous values. If X is to be High or 1 then the value in (IN) should be added to the previous to give a total value out.

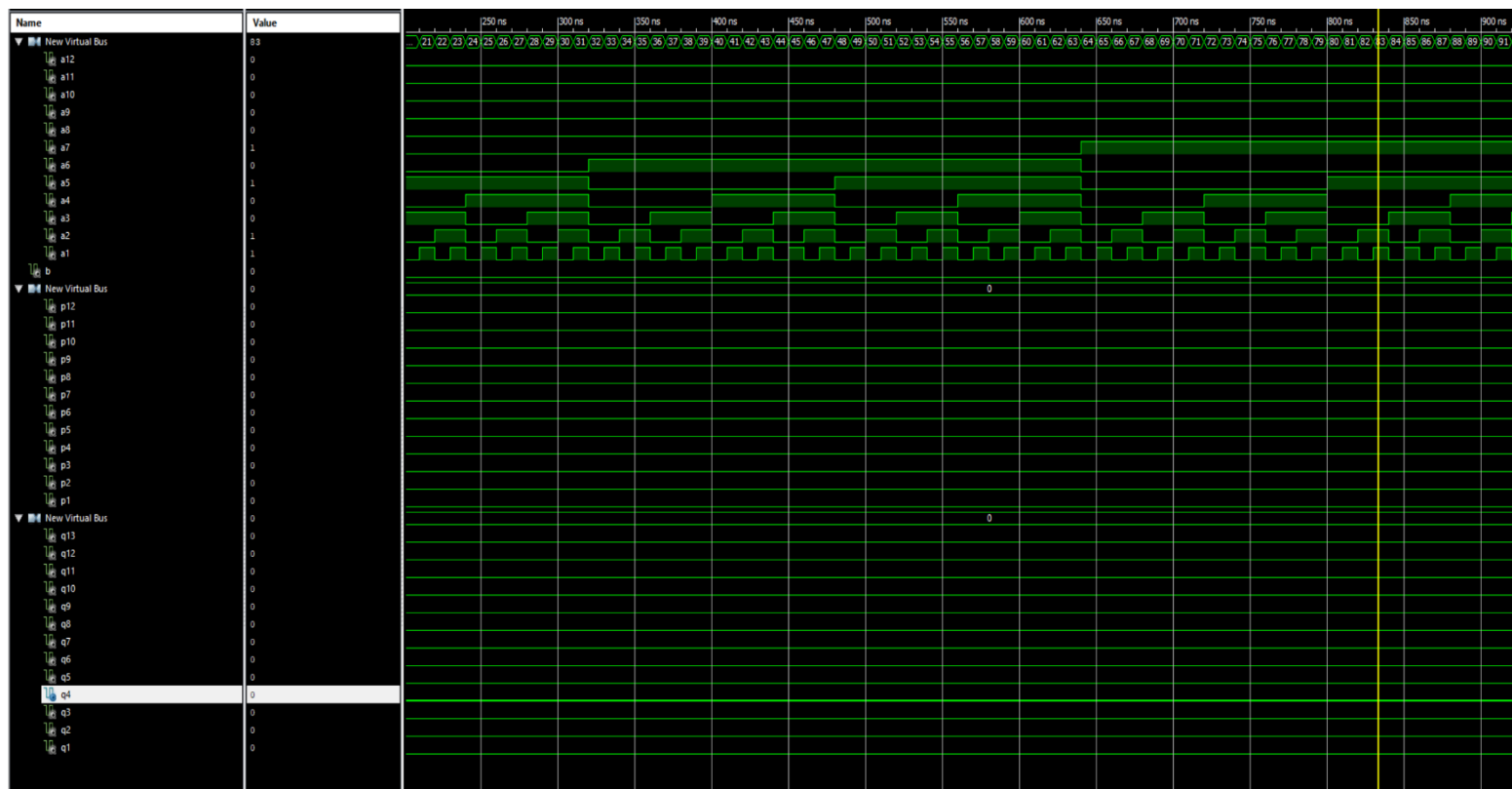


Figure 32 – LineX Wave form

The below figure 31 shows the whole system map out as a Brain Multiplier, this gives a much better visual representation of the Braun multiplier system by carrying the next line across.

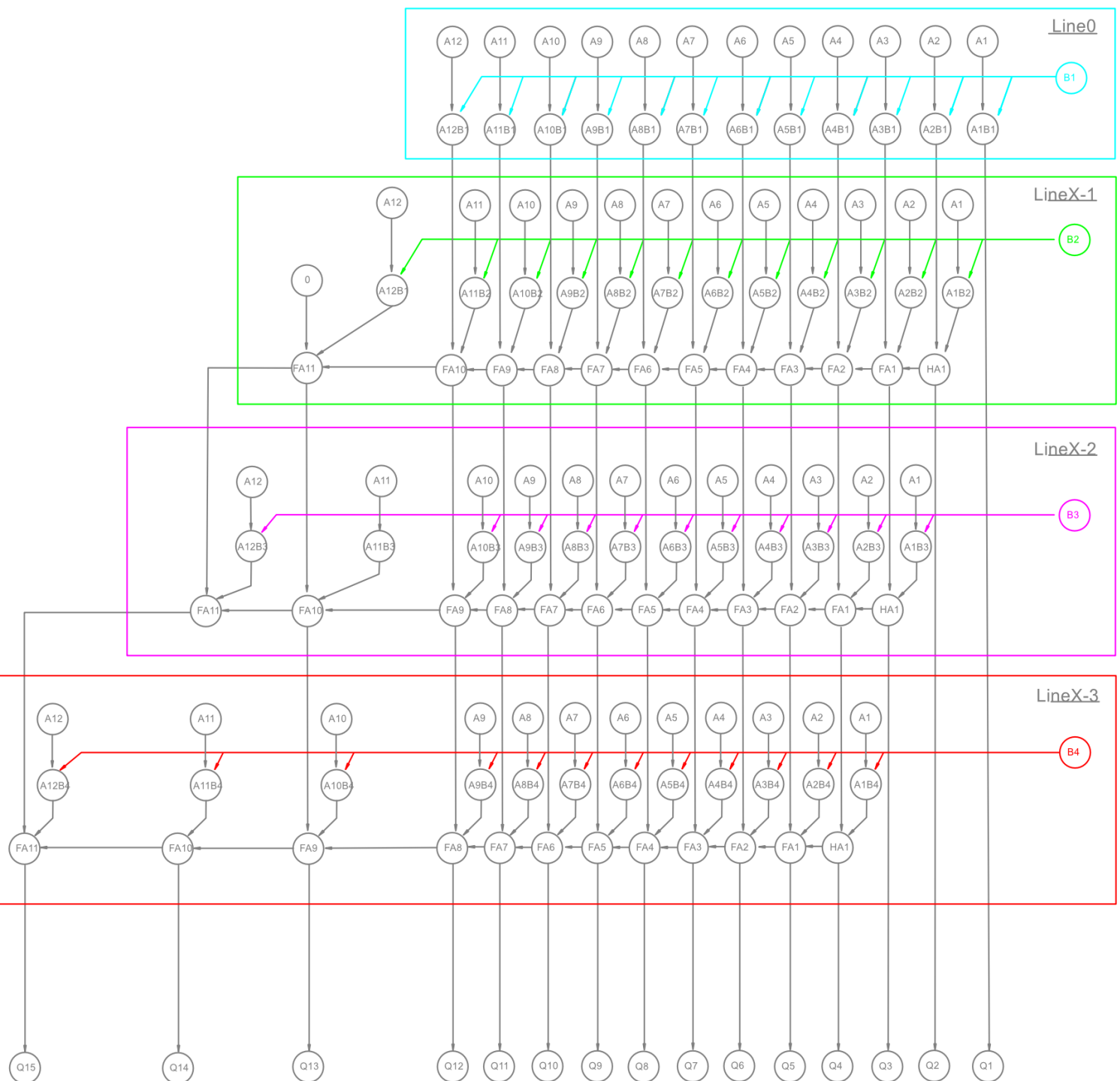


Figure 33 – System over view as Braun multiplier diagram

Design, Simulate and Synthesis A FSM

Within this section the requirement is to design, simulate and synthesise a VHDL code, it should also be able to demon state how an FSM can be designed built and simulated using a discrete gate solution. The design will be based on figure 32 shown below (state diagram).

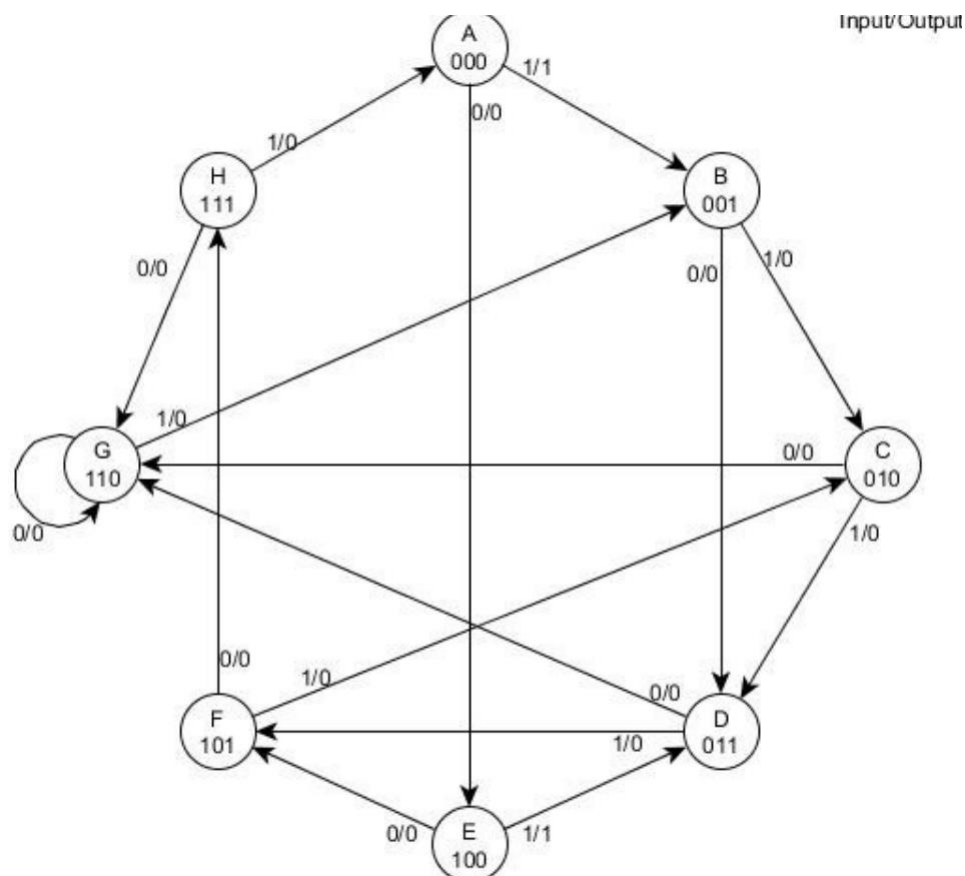


Figure 32 – state diagram for assignment brief

The FSM that will be designed in a discrete gate solution as per the figure 31, it is a synchronous sequential circuit that is governed by clock signals, 1 bit per clock pulse. The circuit has a single input that is either a 1 or 0.

Next state table

As per the state diagram (Figure 32), the current state and next state table can be created. The below state and next state table show the sequence of behaviour of the circuit.

Current State	A	B	C	X
A	0	0	0	0
	0	0	0	1
B	0	0	1	0
	0	0	1	1
C	0	1	1	0
	0	1	1	1
D	0	1	1	0
	0	1	1	1
E	1	0	1	0
	1	0	1	1
F	1	0	1	0
	1	0	1	1
H	1	1	0	0
	1	1	0	1
G	1	1	1	0
	1	1	1	1

Figure 33 – Current State Table

Next State	A	B	C	Y
A	1	0	0	
	0	0	1	
B	0	0	1	1
	0	1	0	
C	1	1	0	
	0	1	1	
D	1	1	0	
	1	0	1	
E	1	0	1	1
	0	1	1	
F	1	1	1	
	0	1	0	
H	1	1	0	
	0	0	1	
G	1	1	0	
	0	0	0	

Figure 34 – Current State Table

Karnaugh Map's

Maurice Karnaugh, a telecommunications engineers, developed the Karnaugh map at Bell Labs in 1953 while designing digital logic based telephone switching circuits. [5]

Karnaugh Maps help by reducing the overall function length by combining longer codes into smaller groups.

Da	ab				
cx	Da	00	01	11	10
	01	1	1	1	1
	01	0	0	0	0
	11	0	1	0	0
	10	0	1	0	1

$$\underline{Da = C.X + / A.B.C + A./B./X}$$

Db	ab				
cx	Da	00	01	11	10
	01	0	1	1	0
	01	0	1	0	1
	11	1	0	0	1
	10	1	1	1	1

$$\underline{Db = C.X + A./B.X + / A.B./C + B./C/X + /A./B.C}$$

Dc	ab				
cx	Da	00	01	11	10
	01	0	0	0	1
	01	1	1	1	1
	11	0	1	0	0
	10	1	0	0	1

$$\underline{Dc = /C.X + A./B./X + / A.B.X + / A./B.C./X}$$

References

- 1 - https://en.wikipedia.org/wiki/Binary_multiplier
- 2 - <https://learn.saylor.org/mod/page/view.php?id=27115&forceview=1>
- 3- <https://www.geeksforgeeks.org/computer-organization-booths-algorithm/>
- 4 - <https://core.ac.uk/download/pdf/234643291.pdf>
- 5 - <https://www.allaboutcircuits.com/textbook/digital/chpt-8/karnaugh-maps-truth-tables-boolean-expressions/>